

April 2022

## **Malware and Memory Forensics on M1 Macs**

Charles E. Glass

*Louisiana State University and Agricultural and Mechanical College*

Follow this and additional works at: [https://repository.lsu.edu/gradschool\\_theses](https://repository.lsu.edu/gradschool_theses)



Part of the [Information Security Commons](#), [OS and Networks Commons](#), and the [Other Computer Sciences Commons](#)

---

### **Recommended Citation**

Glass, Charles E., "Malware and Memory Forensics on M1 Macs" (2022). *LSU Master's Theses*. 5557.  
[https://repository.lsu.edu/gradschool\\_theses/5557](https://repository.lsu.edu/gradschool_theses/5557)

This Thesis is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact [gradetd@lsu.edu](mailto:gradetd@lsu.edu).

# MALWARE AND MEMORY FORENSICS ON M1 MACS

A Thesis

Submitted to the Graduate Faculty of the  
Louisiana State University and  
Agricultural and Mechanical College  
in partial fulfillment of the  
requirements for the degree of  
Master of Science

in

The Department of Computer Science and Engineering

by  
Charles Elliot Glass  
B.S., Louisiana State University, 2019  
May 2022

## Acknowledgments

First and foremost, it is necessary to thank Dr. Golden Richard III for continued guidance throughout time spent at LSU as both an undergraduate and graduate student. Andrew Case has also been a steady source of direction and a wealth of knowledge to call upon during this research. Dr. Gerald Baumgartner and Dr. Anas Mahmoud, for being on the thesis committee. For the Objective-See blog, the Objective by the Sea conference, and the Art of Mac Malware book, Patrick Wardle's contributions to the macOS security space, and thus this research, cannot be understated. Finally, the Glass family for providing the stable base from which this research could develop. It is not possible to make this section comprehensive, so to everyone else who has played a part, large or small, thank you.

# Table of Contents

Acknowledgements . . . . .	ii
List of Tables . . . . .	v
List of Figures . . . . .	vi
Abstract . . . . .	vii
Chapter 1. Introduction . . . . .	1
1.1. Memory Forensics . . . . .	1
1.2. Rosetta 2 . . . . .	3
1.3. Research Importance . . . . .	5
1.4. Outline . . . . .	5
Chapter 2. Related Work . . . . .	7
2.1. M1 Forensic Analysis . . . . .	7
2.2. Memory Analysis on Intel Macs . . . . .	7
2.3. Malware Analysis on Intel Macs . . . . .	8
2.4. Malware on M1 . . . . .	8
Chapter 3. Experimental Setup . . . . .	9
3.1. Environment Setup . . . . .	9
3.2. Malware Selection . . . . .	10
3.3. Forensic Analysis of Malware . . . . .	10
3.4. Acquisition of Memory Image . . . . .	11
Chapter 4. Malware Functionality . . . . .	13
4.1. macOS Security Mechanisms . . . . .	13
4.2. Malware That Was Functional . . . . .	14
4.3. Malware That Was Not Functional . . . . .	23
4.4. Summary . . . . .	28
4.5. Discussion . . . . .	29
Chapter 5. Memory Forensics . . . . .	32
5.1. Background . . . . .	32
5.2. Malware chosen . . . . .	32
5.3. Memory Analysis . . . . .	32

5.4. Analysis Discussion . . . . .	35
Chapter 6. Other Contributions . . . . .	36
6.1. Volatility . . . . .	36
6.2. Volatility on M1 Macs . . . . .	36
6.3. Rosetta 2 Analysis and Plugins . . . . .	37
Chapter 7. Conclusion and Future Work . . . . .	39
7.1. Conclusion . . . . .	39
7.2. Future Work . . . . .	39
References . . . . .	41
Vita . . . . .	46

## List of Tables

4.1.	The functional malware and the protections that were overridden . . . . .	28
4.2.	The non-functional malware and the protections that were overridden . . . . .	28
4.3.	The functional malware and observed high-level behavior . . . . .	29

## List of Figures

1.1.	<i>psscan</i> and other plugins finding a malicious process attempting to hide .....	2
3.1.	Setting Recovery Boot Mode requires SIP to be disabled .....	11
4.1.	<i>CDDS</i> ensuring persistence through a Launch Agent .....	15
4.2.	<i>UserAgent</i> running via a Launch Agent with arguments specified .....	15
4.3.	<i>EvilQuest</i> file infection and persistence .....	17
4.4.	<i>MacDownloader</i> 's harvested info .....	19
4.5.	<i>Realtime-Spy</i> attempting to screen record .....	20
4.6.	<i>ZuRu</i> crashing due to an incompatible dependency .....	22
4.7.	<i>Crisis</i> incompatibility .....	24
4.8.	UDP stream for <i>Silver Sparrow</i> 's installer package .....	26
5.1.	<i>mac_pstree</i> output identifying relationship between <i>relaunch</i> and <i>rtcfg</i> .....	33
5.2.	<i>mac_lsolf</i> output for known malicious PIDs .....	33
5.3.	Artifacts from Rosetta 2 translation for the <i>CDDS</i> malware sample .....	34
5.4.	<i>mac_procdump</i> output for the <i>CarbonComponentScannerXPC</i> process .....	35
6.1.	<i>vmmap</i> showing dual mappings for the test program and library .....	37
6.2.	<i>mac_rosetta</i> extracting AOT artifacts .....	38

## Abstract

As malware continues to evolve, infection mechanisms that can only be seen in memory are increasingly commonplace. These techniques evade traditional forensic analysis, requiring the use of memory forensics. Memory forensics allows for the recovery of historical data created by running malware, including information that it tries to hide. Memory analysis capabilities have lagged behind on Apple's new M1 architecture while the number of malicious programs only grows. To make matters worse, Apple has developed Rosetta 2, the translation layer for running x86\_64 binaries on an M1 Mac. As a result, all malware compiled for Intel Macs is theoretically functional on M1 machines. In this paper, malware will be executed through the Rosetta 2 translation environment in an effort to document the functionality of malware run through it. Afterwards, memory forensics will be performed on select samples to confirm functionality. Finally, the research efforts to bring memory forensics to the M1 will be discussed, along with new artifacts from Rosetta 2 that can be analyzed as a result of these research efforts.



## Chapter 1. Introduction

### 1.1. Memory Forensics

Memory forensics is the examination of a host through the acquisition and analysis of volatile memory. Acquisition of memory samples is possible through a number of tools, many of which are freely available [11] [48] [68]. These tools often work similarly, by loading the kernel driver and determining the ranges of the RAM's physical address space, then translating physical memory to virtual memory through the use of page tables and copying the result to storage. It is also possible to acquire samples through virtual machines, either by suspending the state of the machine or by taking a snapshot in a virtual machine.

After acquisition, parsing is performed by frameworks that present the acquired memory sample from the host in a meaningful state for analysis. To parse memory samples, these frameworks implement strategies like enumerating linked lists and carving, which is the process of searching for objects that look like files, processes, threads, and more. These strategies result in useful structures and objects being exposed from the OS and applications, often even if they were deallocated or terminated prior to acquisition. These structures are used by analysts, who then examine the state of the host through various plugins and tools to look for evidence from an incident.

Memory forensics is an increasingly important part of an incident response team's efforts to understand an incident forensically. It is a necessary addition to traditional storage forensics, as it allows for the recovery and study of information only available in RAM, such as running and hidden processes, open files, encryption keys, private browsing data, network connections, clipboard data, and kernel and application structures. The ability to get information about deallocated objects and terminated processes is extremely useful. It makes it possible to build timelines that would otherwise be incomplete and find indicators of compromise that would otherwise not be seen.

Memory forensics is also important for malware analysis. Through memory forensics,

an analyst can not only see all of the previously mentioned indicators as they relate to the malware, but also defeat many attempts the malware can use to hide itself from traditional analysis and live forensics tools. An example is Direct Kernel Object Manipulation (DKOM), where a rootkit modifies process lists in kernel memory. This modification will allow a process to hide not only from live analysis and OS subsystems, but also from some memory forensics tools. This technique can be seen in action in Figure 1.1, where a malicious application has removed itself from the process list via manipulation of the `_EPROCESS` kernel structure, which is used by the `pplist` plugin and column in that figure. However, it is still detected as active (`TRUE` in the figure) through the Volatility `psscan` plugin. That plugin scans the memory dump for `_EPROCESS` objects using an associated tag, rather than relying on the process list; this method is slower than walking a list, but more thorough.

Offset(P)	Name	PID	pplist	psscan	thrdproc	pspcid	csrss	session	deskthrd	ExitTime
0x0a720180	prtk_worker_ser	360	True	True	False	True	True	True	False	
0x097e0c68	vmtoolsd.exe	3932	True	True	False	True	True	True	True	
0x0a4e0da0	InstallShield L	1292	True	True	False	True	True	True	True	
0x0aad9888	services.exe	1152	True	True	False	True	True	True	True	
0x09852490	wcescomm.exe	4052	True	True	False	True	True	True	True	
0x09ec2da0	wmiprvse.exe	2984	True	True	False	True	True	True	True	
0x09f15810	wdfmgr.exe	2316	True	True	False	True	True	True	True	
0x09f32960	sshd.exe	2080	True	True	False	True	True	True	True	
0x09b49020	CodeMeterCC.exe	1372	True	True	False	True	True	True	True	
0x097a6c28	cmd.exe	2164	True	True	False	True	True	True	True	
0x09f2e518	oracle.exe	2096	True	True	False	True	True	True	True	
0x098185a8	rundll32.exe	3840	True	True	False	True	True	True	True	
0x0a926da0	vmacthlp.exe	1332	True	True	False	True	True	True	True	
0x0a86bb20	svchost.exe	1228	True	True	False	True	True	True	True	
0x0a785da0	winlogon.exe	1056	True	True	False	True	True	True	True	
0x0938f370	GoogleUpdate.ex	1104	True	True	False	True	True	True	True	
0x09f30020	ctfmon.exe	3952	True	True	False	True	True	True	True	
0x0aad9608	svchost.exe	1628	True	True	False	True	True	True	True	
0x09f49da0	cygrunsrv.exe	1596	True	True	False	True	True	True	True	
0x0a607228	CodeMeter.exe	436	True	True	False	True	True	True	True	
0x09f32020	extjob.exe	2072	True	True	False	True	True	True	True	
0x0a509b30	vmtoolsd.exe	2476	True	True	False	True	True	True	True	
0x0a841a88	svchost.exe	1744	True	True	False	True	True	True	True	
0x0a640da0	svchost.exe	1432	True	True	False	True	True	True	True	
0x09f14020	svchost.exe	2240	True	True	False	True	True	True	True	
0x0aad947b8	svchost.exe	396	True	True	False	True	True	True	True	
0x0a70eda0	svchost.exe	504	True	True	False	True	True	True	True	
0x09796020	MagicDisc.exe	2260	False	True	False	True	True	True	True	
0x09cd0028	unauclt.exe	2776	True	True	False	True	True	True	True	
0x097f7020	AdobeARM.exe	3804	True	True	False	True	True	True	True	
0x0a5fdda0	lsass.exe	1164	True	True	False	True	True	True	True	
0x09e4d020	BTTray.exe	2880	True	True	False	True	True	True	True	

Figure 1.1. `psscan` and other plugins finding a malicious process attempting to hide

Plugins that use alternative methods to find processes are also shown, many of which also find the hidden process. Other malicious techniques like DKOM are increasingly commonplace, including memory-only loading, dropping, and execution. Memory forensics is therefore a crucial component, and sometimes the only solution, for a digital forensics investigation.

## **1.2. Rosetta 2**

### **1.2.1. Background and Translation Pipelines**

Rosetta 2 bridges previous-generation x86 Intel apps with M1 Macs. It is translation software that automatically converts an executable with only x86\_64 instructions to arm64, then runs the translated binary [1]. As a result, it is usually the case that the first launch will be slower while translation occurs, but subsequent launches will have only slight reduction in performance compared to running the same binary on an Intel Mac. This method, called ahead-of-time (AOT) translation by Apple, is better than acting simply as a full emulator for x86 when a program is run, as that would noticeably hinder speed and the user experience every time. It is accomplished by generating a Mach-O file with the file extension *.aot*. It is based on the original x86\_64 executable, and only generated once. Whenever the x86\_64 binary needs to be run, this translated AOT file is mapped into the memory of the executing process. There are security mechanisms in place to ensure the translation artifact has not been tampered with, such as code directory hashes and protection through Security Integrity Protection (SIP) [24].

The other translation pipeline for Rosetta 2 is called just-in-time (JIT). JIT is necessary because some applications generate code at runtime. In the case of an x86\_64 application that does this, Rosetta 2 will encounter code that cannot be translated ahead of time. The JIT translation pipeline takes care of these instances entirely within the process after being transferred control by the kernel.

The typical flow for an x86\_64 binary is that the AOT pipeline will first create an

AOT artifact derived from the binary. That artifact is then located when running the app and mapped onto the executing process's memory. JIT is then used for any x86\_64 Mach objects encountered throughout the execution. Constants are referenced in the original binary, not copied into the translated one. Thus, the original binaries are still required; the AOT artifact is not a standalone that can be moved and executed by itself [44].

### 1.2.2. Constraints

Rosetta 2 cannot translate kernel extensions or virtual machine apps that virtualize x86\_64 platforms, but it can translate almost anything else compiled for x86\_64 Macs. It does not come pre-installed on M1 Macs, so the first time Rosetta 2 is required, a prompt to download it appears. It can also be installed through the command line at any time [9]. A final constraint is that Rosetta translation covers the entire process, meaning all code or dependencies loaded by the process must also be x86\_64 if the process was started as such [1].

### 1.2.3. Handling of Mach-O Universal Binaries

The Mach object file format is used as the native format for binaries in macOS, including executables, libraries, and dynamically-loaded code [20]. Mach-O binaries have multiple formats, one of which is universal. Universal binaries support multiple architectures within one Mach-O file, including 32-bit PowerPC, 64-bit PowerPC, i386, x86\_64, and arm64 [14]. For universal executables with both x86\_64 and arm64 code, Rosetta 2 is by default not used at all. The code slice that most closely resembles the architecture of the host will be run, so the arm64 binary will execute. An exception to this behavior is that script-only apps are run under Rosetta 2 translation by default [3]. The user can also decide to execute the x86\_64 slice through Rosetta 2 if, for example, it is expected to load x86\_64 code modules rather than arm64 versions of them. It is additionally possible to specify that an app be run under x86\_64 by default using the *LSArchitecturePriority* key [12]. Another consideration is that running an unsigned universal binary from an arm64

shell will result in the process being killed, due to arm64 executables needing to be signed to run [40]. This constraint does not apply to x86\_64 binaries, which are allowed to run completely unsigned [23].

### **1.3. Research Importance**

It may initially seem like an obvious decision for malicious programmers to want to build universal binaries, so that their malware immediately natively runs as arm64 on M1 Macs. However, there are significant factors discussed in the previous section, such as needing to build all dependencies universally, needing to sign validly to run arm64 binaries at all, and not being able to debug arm64 slices without an M1 Mac themselves. As a result of these constraints, malware authors may prefer building only for x86\_64 and letting Rosetta 2 handle the rest. This possibility makes it even more important to examine the Rosetta 2 translation software and the artifacts it generates.

Last year, data breaches cost companies an average of over \$4 million [6]. Ransomware attacks exceeded \$600 million in 2021, over double the amount in 2020 [61]. As many businesses and livelihoods have shifted online as a result Covid-19, it is more important than ever that digital forensics analysts have the tools they need to fully investigate and remediate incidents. They currently do not on Apple M1 Macs.

In addition, malware is already being compiled with code for M1. Some of these strains are highly infectious, with one sample affecting 30,000 macOS machines [65]. Also, modern malware with memory-only capabilities are known to run on M1 through Rosetta 2. EvilQuest is an example of malware that is tested in this research that attempts in-memory execution of a payload [56]. This example alone shows that there are aspects of malicious attacks that cannot currently be detected on M1 Macs.

### **1.4. Outline**

Chapter 2 will introduce prior works that are related to my research. Chapter 3 will go into detail regarding how the testing environment was set up, how malware samples

were chosen and obtained, the methodology for performing analysis on those samples in Chapter 4, and how memory samples were acquired for the analysis performed in Chapter 5. Chapter 4 will present and discuss the results of executing and observing malware samples in the M1 virtual machine environments. Chapter 5 will discuss the malware selected for further analysis, examine those samples through memory forensics, and finally discuss the results. Chapter 6 will present further research that was performed to enable and enhance memory forensics on Apple M1 Macs. Chapter 7 will provide concluding thoughts on the research presented, and also discusses future work that could be done to build upon this work and M1 memory forensics.

## Chapter 2. Related Work

### 2.1. M1 Forensic Analysis

The research regarding M1 Mac analysis is currently quite sparse. Koh M. Nakagawa has examined M1 Macs, specifically looking at reversing Rosetta 2 [44]. His motivation was to investigate it as an attack surface and potentially exploit it. In the course of his research, he patched Ghidra to correctly disassemble AOT files, reverse engineered the Rosetta 2 runtime binary, detailed how to debug the emulation process at the arm64 instruction level, and more.

The Corellium team have analyzed M1 devices at the hardware level. They did this in an effort to gain knowledge regarding Apple Silicon and get Linux running on it. They have presented their reverse engineering research at BlackHat 2021, where they succeeded in running Linux on M1 hardware [62]. This effort involved the understanding and programming of many Apple hardware tricks, such as adding Apple’s custom interrupt routing support to the Linux kernel [30].

Joshua Duke has also performed relevant research [42], where the full Volatility suite of plugins was run and the output compared between Intel Macs and M1 Macs. Technical explanations for differences were given, and the future necessary fixes for plugins that broke were documented. Results from this work were used to direct the efforts necessary to bring memory forensics to the M1.

### 2.2. Memory Analysis on Intel Macs

Modern macOS Userland Runtime Analysis [51] and Memory Analysis of macOS Page Queues [33] are two examples of recent research that performed memory analysis on macOS to better understand internals and find forensically significant artifacts. The former focused on macOS subsystem and runtime memory analysis capabilities in userland, and it also tested malware samples and their ability to be detected in memory using this analysis. The

latter studied physical page management in an effort to increase the number of artifacts found in macOS memory.

The Art of Memory Forensics [45] is the flagship book on memory forensics. It has over 100 pages dedicated solely to Mac memory forensics, ranging from foundational skills to advanced threat detection.

The Volatility Foundation [63] and Volexity [27] have been a part of advancing memory forensics capabilities through memory analysis for many years, including on Macs [32].

### **2.3. Malware Analysis on Intel Macs**

Objective by the Sea is a Mac-focused cybersecurity conference where presentations, often about malware, are given yearly [18]. Objective-See is a blog maintained by Patrick Wardle, where technical analyses of macOS malware are regularly posted [19]. Patrick Wardle also has a work-in-progress book titled The Art of Mac Malware that covers all aspects of macOS malware and the tools needed to perform analysis [69].

Trendmicro has compiled a detailed technical report on the *XCSSET* macOS malware, including analysis of the binaries and infection mechanisms, as well as reverse-engineering [29]. Many other threat intelligence blogs perform similar analyses regularly [13] [22] [16] [28] [25].

### **2.4. Malware on M1**

As the number of samples grows, there are increasingly many reports on specific samples of malware with native arm64 code.

Of note is Red Canary's *Silver Sparrow* writeup, which exposed and detailed the most infectious strand of M1 malware currently known [66]. Google's Threat Analysis Group discovered a watering hole campaign that included a binary for M1 and targeted visitors to pro-democracy Hong Kong websites [37]. Patrick Wardle also has given a talk at Objective by the Sea v4.0, where he reverse engineered the anti-analysis arm logic of M1 malware found in the wild [58].



## Chapter 3. Experimental Setup

This section will discuss the steps taken to create an environment for safe and thorough testing of malware. It will also discuss how the malware samples were chosen and obtained, as well as how samples were analyzed to determine functionality through Rosetta 2 translation. Finally, it will detail the process to acquire a memory image for examination, as this process on M1 Macs is currently more difficult than with other systems.

### 3.1. Environment Setup

Excluding *EvilQuest*, the malware samples tested for Chapter 4 were executed on virtual machines (VMs) running on Parallels Desktop Pro 17.1.1-51537. The operating system of the M1 Mac VMs was macOS Monterey 12.2, build 21D49, and the operating system of the host M1 Mac was macOS Monterey 12.2.1, build 21D62. The host that the memory was acquired from was running macOS Monterey 12.1.0, build 21C52, in order for the memory sample to be compatible with an already created Volatility profile. Rosetta 2 had to be installed on each VM prior to execution of malware. Command Line Developer Tools were installed as well, as some malware utilized functionality from tools in this suite. On the VMs, Wireshark and various built-in Terminal tools such as *ps aux* were used to monitor high-level functionality of the malware. Each VM was within what Parallels calls a Host-Only Network [55], where the VM is in a separate subnet that can only connect to a gateway and other VMs in the subnet. Each VM was also set up such that it was the DNS Server under its networking configuration. This setup allowed the observation of DNS queries made during dynamic analysis of the malware. Because there is currently no snapshot functionality on M1 images for Parallels, multiple samples were run on one VM before preparing and using another VM for more samples. For all VMs, System Integrity Protection (SIP) remained enabled. As a result, a few samples identified by macOS as malware were unable to run on the VMs. SIP would need to be disabled because the ability

to disable those protection measures is restricted under it. Samples that were applications could still be run by overriding malware protection, but executables, disk images, and package installers could not. It was not possible to disable SIP on a VM as Recovery Mode was not accessible. The reason Recovery Mode was inaccessible is discussed in Section 3.4.

Three malware samples were also run on the host machine while it was isolated from the internet, so that a memory image could be collected for analysis. Section 3.4 also discusses why this was necessary. The host machine was wiped and rebuilt after the memory dump was acquired and moved off disk to a flash drive.

### **3.2. Malware Selection**

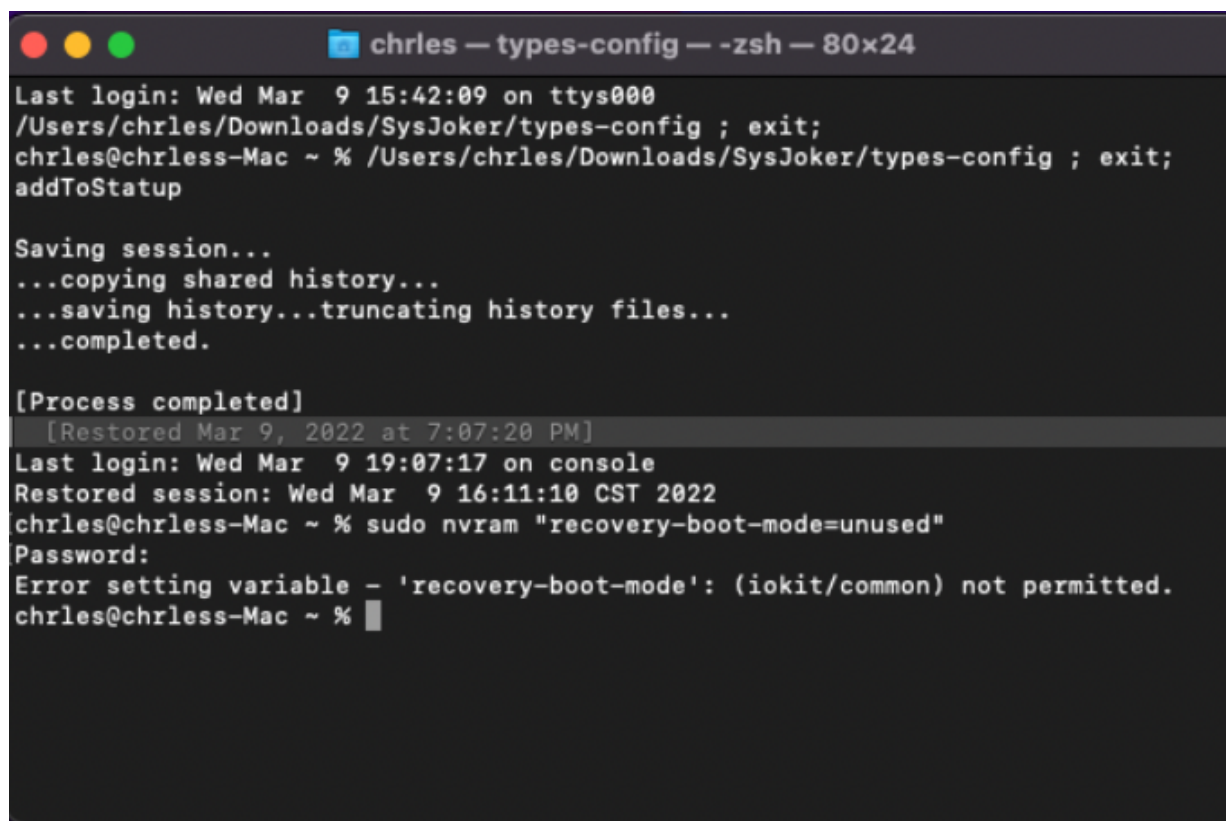
Malware was chosen based on a number of factors, such as impact level, viability of attack vector, and indicators of compromise. Newer malware was also generally given preference to similar malware that was older. For example, *CDDS / MacMa* had both a 2019 and 2021 installer; the 2021 version was the version tested. A majority of samples are thus modern, higher impact malware like backdoors or spyware with persistence mechanisms. In total, roughly 150 malware samples were considered for testing. Those chosen were obtained from the Objective-See blog [15], where many malicious macOS samples dating as far back as the early 2000's are freely available.

### **3.3. Forensic Analysis of Malware**

A high-level analysis was performed for each of the selected 16 malware samples to determine which of them functioned through Rosetta 2 translation and which did not. Samples for this portion were run in the VMs, and clear indicators of compromise, such as Command and Control (C&C) pings or running Launch Agents, were used to determine functionality. For those samples that did not have expected behavior, more analysis was done to determine the cause and whether Rosetta 2 was responsible.

### 3.4. Acquisition of Memory Image

For the malware samples that were chosen for further analysis, a memory sample needed to be obtained. Version 17.1.1-51537 of Parallels Desktop Pro does not support snapshots or suspension of an M1 VM [54], so the virtualization methods of obtaining a memory sample were not possible. In that version of Parallels, boot order configuration options are also not supported. It would be necessary to be able to change the boot order in order to get to Recovery Mode on an M1 VM. If SIP were disabled, it would still be possible via setting Recovery Boot Mode in the terminal. However, in order to disable SIP a user needs to boot into Recovery Mode. Fig 3.1 shows this catch-22.

A terminal window titled "chrles — types-config — -zsh — 80x24" showing a sequence of commands and their outputs. The user runs a script that saves the session and history. After a session restore, the user attempts to run "sudo nvram 'recovery-boot-mode=unused'", which fails with the error "Error setting variable - 'recovery-boot-mode': (iokit/common) not permitted." The terminal text is as follows:

```
Last login: Wed Mar 9 15:42:09 on ttys000
/Users/chrles/Downloads/SysJoker/types-config ; exit;
chrles@chrless-Mac ~ % /Users/chrles/Downloads/SysJoker/types-config ; exit;
addToStatup

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
[Restored Mar 9, 2022 at 7:07:20 PM]
Last login: Wed Mar 9 19:07:17 on console
Restored session: Wed Mar 9 16:11:10 CST 2022
chrles@chrless-Mac ~ % sudo nvram "recovery-boot-mode=unused"
Password:
Error setting variable - 'recovery-boot-mode': (iokit/common) not permitted.
chrles@chrless-Mac ~ % █
```

Figure 3.1. Setting Recovery Boot Mode requires SIP to be disabled

Once in Recovery Mode, it would be possible to allow third-party kernel extensions. This allowance is necessary because memory acquisition tools, such as Volatility's Surge Collect Pro software, require the use of their signed kernel extension to function. With

both the virtualization and Recovery Mode routes not possible for acquiring the memory of a VM, the decision was made to run the malware samples on the host M1 with networking disabled. The host was prepared with Surge and the password-protected zip of malware samples prior to disabling networking. With the host prepared, the samples were run and the memory dump was captured. The dump was moved to a USB and the host was restored to a known good state.

## Chapter 4. Malware Functionality

### 4.1. macOS Security Mechanisms

The malware tested in this research encountered multiple built-in macOS security mechanisms. The below results report, in part, which samples were stopped by which mechanisms. Thus, it is useful to introduce them here.

#### 4.1.1. File Quarantine

File quarantine is a security feature where files downloaded from the internet are marked as quarantined. When quarantined files are first run, a user needs to give permission via an opened dialog box before the file executes. Files downloaded via *curl* and some other methods, however, are not marked as quarantined [35], so this method of protecting users is capable of being bypassed.

#### 4.1.2. Gatekeeper

Gatekeeper is a security feature that is intended to prevent apps developed by an unknown or invalid developer from running without additional user consent. This method has been more effective, but is ultimately still susceptible to users ignoring the warning and deciding to run the program anyway. Also, if they decide to, malware authors are often easily able to bypass this mechanism by signing their apps with illegitimately obtained but valid developer IDs [57].

#### 4.1.3. Notarization

Notarization is a security feature that automatically scans quarantined apps for issues related to code-signing or malicious content [17]. Once an app, package installer, or disk image passes the checks, that information is published online for Gatekeeper to check when users download and run it. If a user's system encounters a previously unnotarized program, it will inform the user and then let them bypass and run it if they choose [39].

#### 4.1.4. Malware Protection

As a final security measure, Apple also develops an antivirus called *XProtect*. *XProtect* is built-in to macOS and uses YARA signatures to detect malware and prevent it from running [21]. When a sample was flagged as malware, it either requires overriding malware protection for that app, or disabling of SIP in order to run executables. *EvilQuest* is an executable that did not have the “Override Malware Protection” option, while the *MacDownloader*, *XCSSET*, and *ZuRu* applications did. Therefore, it seems the ability to override malware protections exists for applications but not for executables. This observation was not able to be confirmed through documentation, however.

## 4.2. Malware That Was Functional

### 4.2.1. CDDS / MacMa

#### *Background*

*CDDS / MacMa* was discovered in November 2021 and targeted visitors to a few pro-democracy websites in Hong Kong. It has abilities including execution of terminal commands given remotely, file transfer, audio recording, and more [37].

#### *Execution*

After running the 2021 version of the installer, a payload is dropped that triggers warnings such as attempting to control the computer using accessibility features and accessing the microphone. *UserAgent* is the executable that is dropped and persists via a Launch Agent. Launch Agents are perhaps the most common method for malware to persist on macOS. Launch Agents can be used to take actions whenever directories change, start jobs at set intervals, and even run scripts every time a user logs in [5]. Launch Daemons are largely equivalent, except they run on behalf of the root user or another specified user [10].

As an example of a Launch Agent, *CDDS*'s can be seen in Figure 6.1. *UserAgent* running accordingly is shown in Figure 4.2

```

CDDS — vim ~/Library/LaunchAgents/com.UserAgent.va.plist — 93x26
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.UserAgent.va</string>
  <key>LimitLoadToSessionType</key>
  <string>Aqua</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/chrles/Library/Preferences/UserAgent/lib/UserAgent</string>
    <string>-runMode</string>
    <string>ifneeded</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>StartInterval</key>
  <integer>600</integer>
  <key>ThrottleInterval</key>
  <integer>2</integer>
  <key>WorkingDirectory</key>
  <string>/Users/chrles/Library/Preferences/UserAgent/lib</string>
</dict>
</plist>
~/Library/LaunchAgents/com.UserAgent.va.plist" 24L, 707B

```

Figure 4.1. *CDDS* ensuring persistence through a Launch Agent

```

Downloads — -zsh — 80x24
-      0      com.apple.batteryintelligenced
336    0      com.apple.GameController.gamecontrollerd
554    0      com.apple.AppStoreDaemon.StorePrivilegedTaskService
222    0      com.apple.apfsd
111    0      com.apple.dasd
-      0      com.apple.sessionlogoutd
-      0      com.apple.AssetCacheManagerService
147    0      com.apple.systemstatusd
-      0      com.apple.cmio.AVCAssistant
160    0      com.apple.analyticsd
189    0      com.apple.diskmanagementd
188    0      com.apple.CodeSigningHelper
123    0      com.apple.sandboxd
310    0      com.apple.sysmond
124    0      com.apple.corebrightnessd
-      0      com.apple.tailspind
chrles@chrless-Mac Downloads % ps aux | grep UserAgent
chrles      943  1.2  0.2 34741876  9556  ??  S      2:39PM  0:07.44 /
Users/chrles/Library/Preferences/UserAgent/lib/UserAgent -runMode ifneeded
chrles      345  0.0  0.1 408034224   3088  ??  S      12:04PM  0:00.01
/System/Library/CoreServices/APFSUserAgent
chrles     1012  0.0  0.0 408637536   1696  s000  S+    2:52PM  0:00.00
grep UserAgent
chrles@chrless-Mac Downloads % █

```

Figure 4.2. *UserAgent* running via a Launch Agent with arguments specified

### 4.2.2. DazzleSpy

#### *Background*

*DazzleSpy* was discovered in January 2022 and was distributed via a compromised Hong Kong pro-democracy website. It supports fully interactive RDP sessions, the ability to dump the Keychain on vulnerable systems, and self-deletion [49]. This sample comes from a similar campaign to *CDDS*, with the main difference being the payloads (*CDDS* and *DazzleSpy*) that are delivered.

#### *Execution*

After running *DazzleSpy*, connection attempts to its probable C&C were observed. In addition, a Launch Agent was loaded that will launch the executable each time the user logs in.

### 4.2.3. EvilQuest

#### *Background*

*EvilQuest* was discovered in June 2020 and was distributed predominately through pirated programs. It has the ability to ransom a host, infect files to run its payload, search for and kill processes, and harvest data [56].

#### *Execution*

The malware bundled with the pirated program was identified as such by macOS, so it was prevented from running. As discussed in prior sections, it is not possible to fully disable malware protections for executables with the current version of Parallels for M1 Mac VMs. As this sample was of particular interest due to its distribution vector, capabilities, and relative recency, it was selected for further analysis and run on the host machine before a memory image was collected. Just from initial observation, persistence was set, test binaries were infected, and a fake process named *Avast* was killed. The file infection was marked by the ending hex *0xdeadface* and still maintained functionality of the initial binary. Networking could not be tested because it was disabled on the host, but based on other



functionality it is likely to have attempted to connect to C&C.

Figure 4.3 shows both a running Launch Agent and file infecteded from *EvilQuest*.

```
user@cyber2-2 darwin % ps aux | grep silent
root          1529   0.1  0.0 408666384   7792   ??  Ss   6:38PM   0:00.02
sudo /Library/AppQuest/com.apple.questd --silent
root          1530   0.1  0.0 34260360    2668   ??  S    6:38PM   0:00.01 /
Library/AppQuest/com.apple.questd --silent
user          1533   0.0  0.0 408628368   1616  s000  S+   6:38PM   0:00.00
grep silent
user@cyber2-2 darwin % hexdump ~/Downloads/testBinary | tail -n 10
0021950 65 61 64 65 72 00 5f 68 65 6c 6c 6f 00 5f 6d 61
0021960 69 6e 00 5f 6e 75 6d 00 5f 6e 75 6d 32 00 5f 6e
0021970 75 6d 62 65 72 5f 00 5f 5f 5f 73 74 72 63 70 79
0021980 5f 63 68 6b 00 5f 66 72 65 65 00 5f 67 65 74 63
0021990 68 61 72 00 5f 6d 61 6c 6c 6f 63 00 5f 70 72 69
00219a0 6e 74 66 00 64 79 6c 64 5f 73 74 75 62 5f 62 69
00219b0 6e 64 65 72 00 5f 5f 64 79 6c 64 5f 70 72 69 76
00219c0 61 74 65 00 00 00 00 03 70 57 01 00 ce fa ad
00219d0 de
00219d1
user@cyber2-2 darwin %
```

Figure 4.3. *EvilQuest* file infection and persistence

#### 4.2.4. FinSpy

##### *Background*

The macOS version of *FinSpy* was discovered by Amnesty International in Fall 2019. The original Windows version was distributed through a backdoored version of Adobe update [31] and the macOS version was distributed as a trojaned application with a Turkish name [53]. It is used as spyware.

##### *Execution*

After the disk image was mounted and the installer app was run, a malicious installer was dropped and executed at the same time a network request to download a legitimate version of Adobe AIR was made. The malicious installer then detected it was running inside virtualization software and stopped running. As the focus of this portion of the research was confirming high-level functionality through Rosetta 2 translation, using a debugger to get around this check was outside of scope.

#### 4.2.5. Hydromac

##### *Background*

*Hydromac* was discovered in the middle of 2021 and was distributed via the *Tarmac* and *Bundlore* malwares [46]. It is thought to be a downloader/stager, as it has the ability to check for installed anti-virus programs and can download and execute programs.

##### *Execution*

After giving execution permissions, it runs and attempts to connect out to C&C. No other functionality was observed.

#### 4.2.6. Komplex

##### *Background*

*Komplex* was discovered in late 2016 and was associated with APT28 Sofacy. It likely targeted individuals working in aerospace and was distributed as an attachment in an email. It has functionality including remote execution and information collection about the host it runs on [36].

##### *Execution*

Upon running *Komplex*, a PDF in Russian opened that appeared to relate to a space program. It also dropped and ran an executable that checked for connectivity, as well as creating a Launch Agent that launches that script at startup. It also deleted itself after dropping the other executable. However, an additional script that was supposed to be responsible for loading the Launch Agent did not get created or run, so persistence was ultimately not set. This was confirmed by rebooting the system and noting that the payload did not start running. The VM this malware ran on was isolated from the internet, so the connectivity check failed. Parallels Host-Only mode did not seem to allow a VM to see other VMs on the same subnet, though the documentation implies it should. As a result, using a program like INetSim [8] on another VM to return 200's to the malware did not seem possible and was out of scope since functionality was already observed.

### 4.2.7. MacDownloader

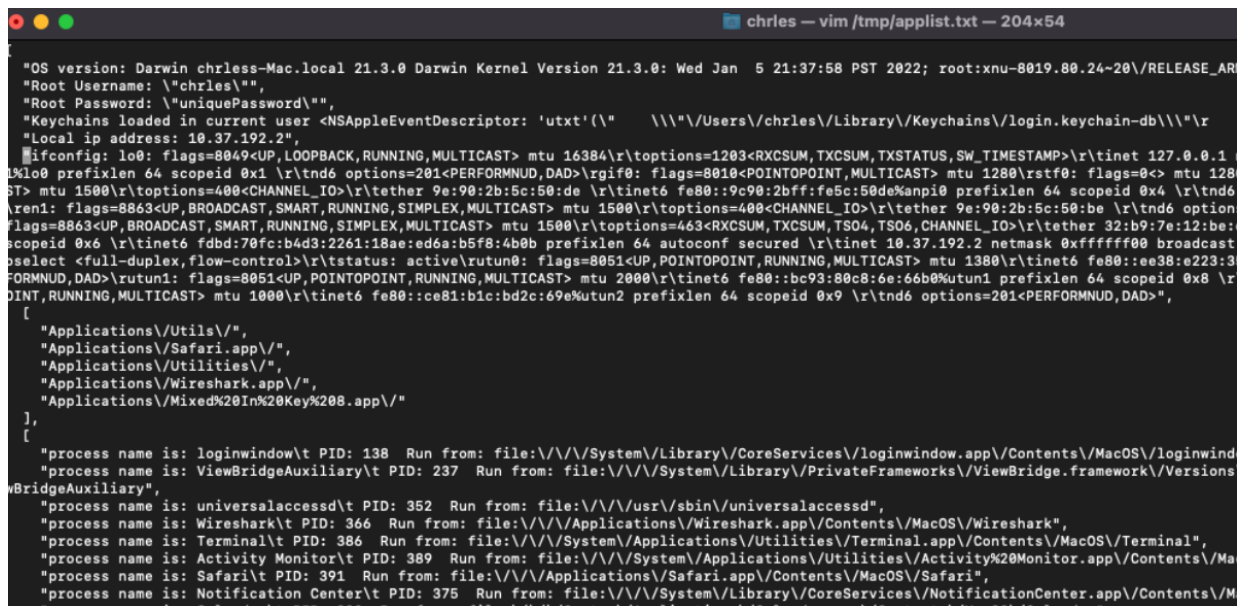
#### Background

*MacDownloader* was discovered in early 2017 and was associated with Iranian APTs Charming Kitten and Flying Kitten. It was distributed as both an Adobe Flash installer and BitDefender adware remover, and has the ability to harvest and upload sensitive information [7].

#### Execution

After running it, it completes a fake installation while asking the user for their credentials. A file is created in the temp directory that is populated with sensitive information, including username and password, Keychain files, running processes, and more. However, traffic was not seen to its known domain.

In Figure 4.4, the harvested info from a *MacDownloader* execution can be seen.



```
OS version: Darwin chrless-Mac.local 21.3.0 Darwin Kernel Version 21.3.0: Wed Jan  5 21:37:58 PST 2022; root:xnu-8019.80.24~20/RELEASE_ARM
"Root Username: \"chrless\"",
"Root Password: \"uniquePassword\"",
"Keychains loaded in current user <NSAppleEventDescriptor: 'utxt'(\"  \\\"\\Users\\chrless\\Library\\Keychains\\login.keychain-db\\\"\\r
"Local ip address: 10.37.192.2",
"ifconfig: lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384\\r\\toptions=1203<RXCSUM, TXCSUM, TXSTATUS, SW_TIMESTAMP>\\r\\tinet 127.0.0.1
%lo0 prefixlen 64 scopeid 0x1 \\r\\tnd6 options=201<PERFORMNUD, DAD>\\r\\gif0: flags=8010<POINTOPOINT, MULTICAST> mtu 1280\\r\\stf0: flags=0<> mtu 128
ST> mtu 1500\\r\\toptions=400<CHANNEL_IO>\\r\\tether 9e:90:2b:5c:50:de \\r\\tinet6 fe80::9c90:2bff:fe5c:50de%anpi0 prefixlen 64 scopeid 0x4 \\r\\tnd6
aren1: flags=8863<UP, BROADCAST, SMART, RUNNING, SIMPLEX, MULTICAST> mtu 1500\\r\\toptions=400<CHANNEL_IO>\\r\\tether 9e:90:2b:5c:50:be \\r\\tnd6 option
flags=8863<UP, BROADCAST, SMART, RUNNING, SIMPLEX, MULTICAST> mtu 1500\\r\\toptions=463<RXCSUM, TXCSUM, TSO4, TSO6, CHANNEL_IO>\\r\\tether 32:b9:7e:12:be:
scopeid 0x6 \\r\\tinet6 fdbd:70fc:b4d3:2261:18ae:ed6a:b5f8:4b0b prefixlen 64 autoconf secured \\r\\tinet 10.37.192.2 netmask 0xfffff00 broadcast
pselect <full-duplex, flow-control>\\r\\tstatus: active\\rutun0: flags=8051<UP, POINTOPOINT, RUNNING, MULTICAST> mtu 1380\\r\\tinet6 fe80::ee38:e223:3
FORMNUD, DAD>\\rutun1: flags=8051<UP, POINTOPOINT, RUNNING, MULTICAST> mtu 2000\\r\\tinet6 fe80::bc93:80c8:6e:66b0%utun1 prefixlen 64 scopeid 0x8 \\r
POINT, RUNNING, MULTICAST> mtu 1000\\r\\tinet6 fe80::ce81:b1c:bd2c:69e%utun2 prefixlen 64 scopeid 0x9 \\r\\tnd6 options=201<PERFORMNUD, DAD>",
[
  "Applications\\Utils\\",
  "Applications\\Safari.app\\",
  "Applications\\Utilities\\",
  "Applications\\Wireshark.app\\",
  "Applications\\Mixed%20In%20Key%208.app\\"
],
[
  "process name is: loginwindow\\t PID: 138 Run from: file:\\\\\\System\\Library\\CoreServices\\loginwindow.app\\Contents\\MacOS\\loginwind
"process name is: ViewBridgeAuxiliary\\t PID: 237 Run from: file:\\\\\\System\\Library\\PrivateFrameworks\\ViewBridge.framework\\Versions
ViewBridgeAuxiliary",
  "process name is: universalaccessd\\t PID: 352 Run from: file:\\\\\\usr\\sbin\\universalaccessd",
  "process name is: Wireshark\\t PID: 366 Run from: file:\\\\\\Applications\\Wireshark.app\\Contents\\MacOS\\Wireshark",
  "process name is: Terminal\\t PID: 386 Run from: file:\\\\\\System\\Applications\\Utilities\\Terminal.app\\Contents\\MacOS\\Terminal",
  "process name is: Activity Monitor\\t PID: 389 Run from: file:\\\\\\System\\Applications\\Utilities\\Activity%20Monitor.app\\Contents\\Ma
"process name is: Safari\\t PID: 391 Run from: file:\\\\\\Applications\\Safari.app\\Contents\\MacOS\\Safari",
  "process name is: Notification Center\\t PID: 375 Run from: file:\\\\\\System\\Library\\CoreServices\\NotificationCenter.app\\Contents\\M
"process name is: Calendar\\t PID: 393 Run from: file:\\\\\\System\\Applications\\Calendar.app\\Contents\\MacOS\\Calendar"
```

Figure 4.4. *MacDownloader*'s harvested info

### 4.2.8. Realtime-Spy

#### Background

This *Realtime-Spy* sample is from early 2018, was distributed via the *realtime-spy-*

mac[.]com website, and has the ability to upload screen recordings, keystrokes, and other user activity it has collected [52].

### Execution

While the sample runs, it can be seen reaching out to its namesake domain and attempting to screen record.

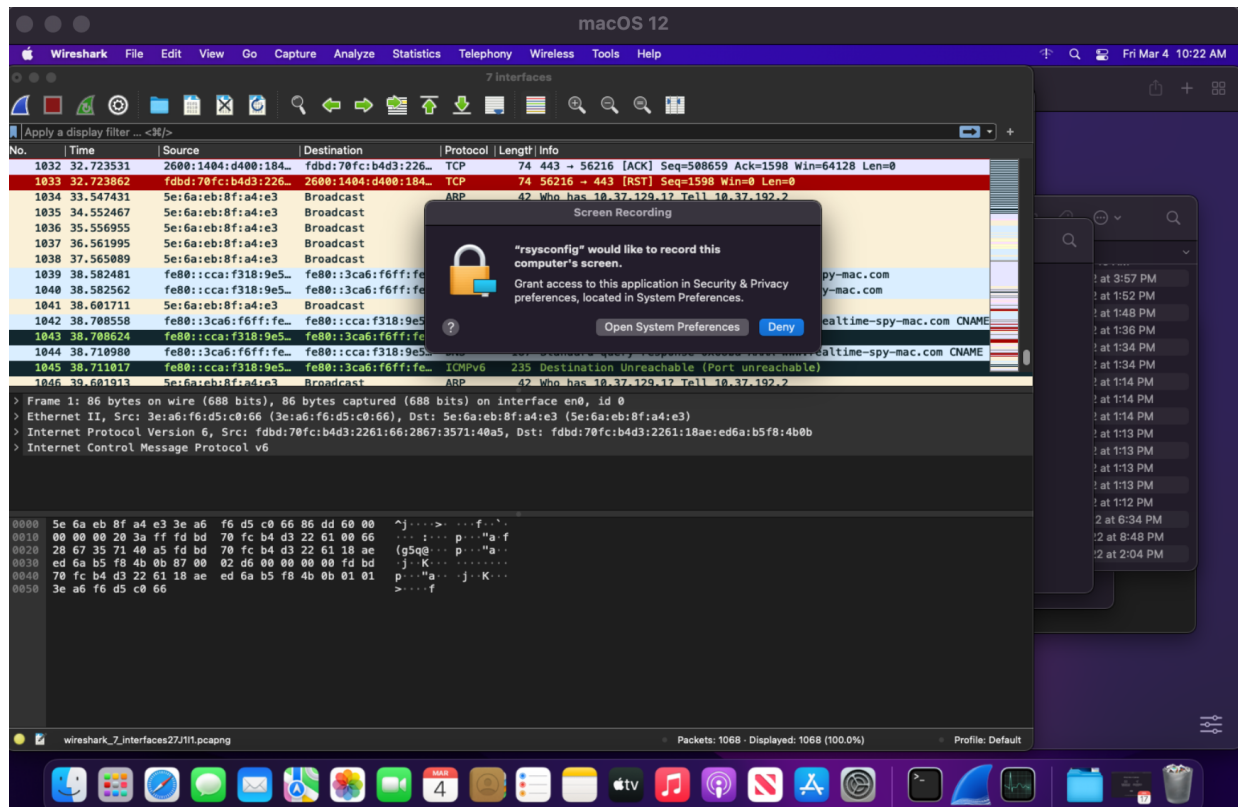


Figure 4.5. *Realtime-Spy* attempting to screen record

## 4.2.9. Ventir

### Background

*Ventir* was discovered back in 2014 and has the ability to execute commands, exfiltrate files, and keylog on a fully infected system [50].

### Execution

It drops 5 binaries and removes itself, creates and loads a launch agent for persistence, and runs one of the dropped binaries. However, the keylogging functionality fails as the

binary responsible is a universal binary compiled for i386 and PowerPC.

#### **4.2.10. XCSSET**

##### *Background*

*XCSSET* was discovered in early or mid 2020, spreads through infected XCode projects, and attempts to steal credentials and data through trojanized browsers [43].

##### *Execution*

Running either of the associated scripts fails, as they are fingerprinted as malware and prevented from executing. Running the infected XCode application after overriding malware protection, however, results in a connection attempt to C&C infrastructure. None of the other infection markers, such as processes being killed or fake apps being created, are observed due to the failed connection attempt.

#### **4.2.11. ZuRu**

##### *Background*

*ZuRu* was discovered in September 2021. It spread through the search engine Baidu, where the authors paid to show at the top of the search results for “iTerm2” via a sponsored link. As a result, it appeared above even the legitimate iTerm2, likely tricking many users. Its functionality includes downloading and executing additional files that exfiltrate information and leverage remote access [59].

##### *Execution*

An initial run was terminated at launch because it was not able to load in a custom malicious dynamically linked library. After investigation, it was determined that the library wasn’t loaded in due to macOS expecting it to be arm64 rather than the x86\_64 library it found. The library was expected to be arm64 because this application was a universal binary, so by default it ran as arm64 on an M1 Mac. After selecting “Open using Rosetta”, it did not crash. It attempted to connect out to C&C but stopped running because it did not receive a response.

Using a Terminal, it is also possible to run the app without the “Open using Rosetta” checkbox selected. After navigating the application’s directory to the actual program, it can be executed through the *arch* command, which allows for specification of the architecture with which to execute a universal binary. Both methods are equivalent and successfully run the application. This successful execution means that Rosetta 2 handles both translation of x86\_64 code and also any x86\_64 dependencies. However, when an app is started as arm64, even if a translatable x86\_64 dependency is found, the app will fail to run. The implication then is that if a malware author is going to build malware as a universal binary, any dependencies within the app will need to be universal as well. This implication is confirmed through Apple documentation. Otherwise, their app will fail on M1 Macs, since arm64 is the default architecture that will run in a universal binary on M1 Macs [2].

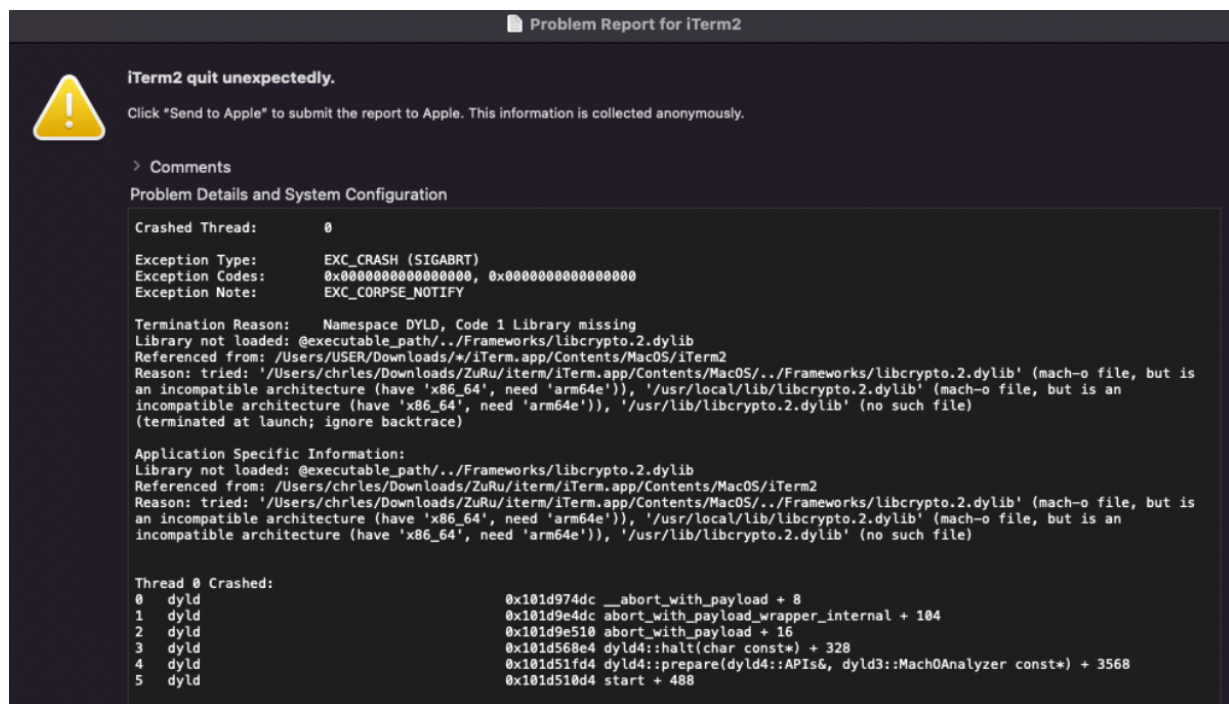


Figure 4.6. *ZuRu* crashing due to an incompatible dependency

As a result, macOS will expect dependencies that a binary loads in to be arm64 as well. The other way to prevent a crash would be for the author to force execution as x86\_64 on an infected host, but at that point there would be no reason to have also built as arm64 in

the first place.

In Figure 4.6, *ZuRu*'s crash report when run as arm64 can be seen. It includes errors regarding incompatible architecture due to finding an x86\_64 dependency it expected to be arm64.

### 4.3. Malware That Was Not Functional

Malware was determined to be nonfunctional if it needed significant modification or to be handheld through bypassing multiple security mechanisms for different stages of its payloads. As an example, *Applejeus* was caught by Gatekeeper, then had a second and third stage get immediately caught by malware protection. Another example, *SysJoker*, needed to have its file extension modified on top of being provided executable permissions to run correctly. This modification changed the entire attack vector of *SysJoker*, as its file extension is thought to play an important part of the initial infection vector, either as a malicious media player attachment or as a TypeScript file inside of an infected package.

#### 4.3.1. Applejeus / MacLoader

##### *Background*

*Applejeus* was discovered around the middle of 2018, is associated with a North Korean APT, was distributed through phishing emails and trojanized programs [38], and has functionality including execution of memory-only payloads.

##### *Execution*

The initial package installer ran, but did not function as intended. It gave root permissions to the malicious executable *unioncryptoupdater* and moved it to a separate location, but that executable was not launched. This failure seems to be due to *unioncryptoupdater* being identified as malware (MACOS.8d038b3) and prevented from running. The same occurs with the *CrashReporter* malware that another *Applejeus* package installer tries to drop and run as a Launch Daemon.

### 4.3.2. Crisis

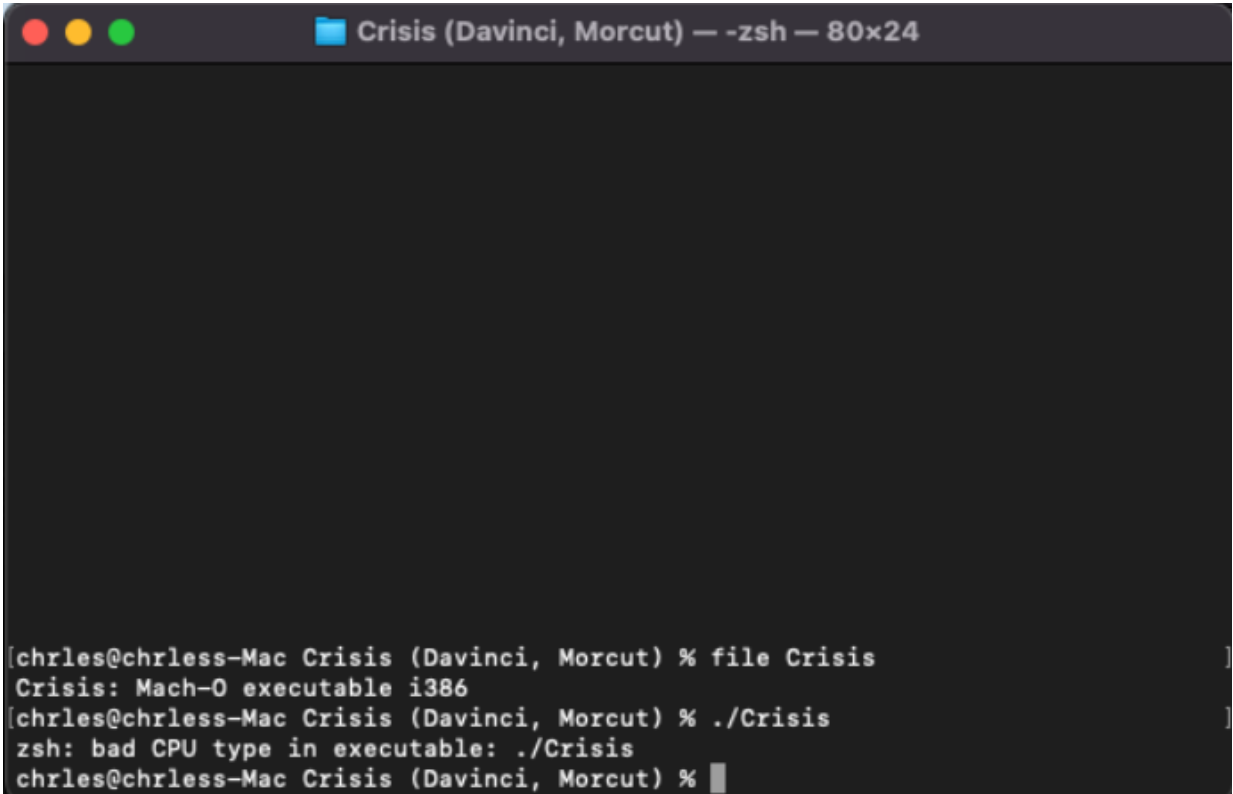
#### *Background*

*Crisis* was discovered in 2016, appeared to be sold as a part of government surveillance tools, and used Adobe-related phishing as an infection method. It spies on users via audio recording, stealing browser data, and more [47].

#### *Execution*

It failed to run due to being a 32-bit (i386) executable, which is incompatible on an M1 even with Rosetta 2 installed.

In Figure 4.7, Rosetta being unable to translate the *Crisis* i386 executable can be seen.



```
Crisis (Davinci, Morcut) — zsh — 80x24

[chrles@chrless-Mac Crisis (Davinci, Morcut) % file Crisis
Crisis: Mach-O executable i386
[chrles@chrless-Mac Crisis (Davinci, Morcut) % ./Crisis
zsh: bad CPU type in executable: ./Crisis
chrles@chrless-Mac Crisis (Davinci, Morcut) % █
```

Figure 4.7. *Crisis* incompatibility



### 4.3.3. Shlayer

#### *Background*

*Shlayer* was originally discovered in 2018, with new variants found through 2021. It is distributed through malicious sites intended to be discovered through search engine optimization, that then redirect to fake Adobe Flash installation pages. A recent variant from last year had the ability to run only with a user double-click, as it utilized 0-days to bypass most Apple security measures described in section 4.1. After infection, it only attempts to download adware [41].

#### *Execution*

The disk image that contains the installer, *AdobeFlashPlayer.dmg*, was flagged as malicious and unable to be mounted. The installer package within, *Player.pkg*, was not flagged as malware and ran, but didn't appear functional at all. This lack of functionality is unsurprising because the malware relied on 0-days that were patched on the OS build it was run on. The ways that the macOS security mechanisms were bypassed were not highly technical, however. They essentially were the result of leaving out an *Info.plist* file of an app bundle that was script-based. These exploits serve as a good reminder that, while M1 Macs seem hardened against malware executing without user permission, it is difficult for a system to be truly secure.

### 4.3.4. Silver Sparrow

#### *Background*

Silver Sparrow was discovered in February 2021, is distributed in installer packages, is a universal binary, and has an unknown final payload. The widest held opinion is that the second stage installs adware, based on similarities between data from the C&C that includes the arg *upbuchupsf*, and an affiliate code often used by adware. It also has roughly 30,000 detections on macOS [66].

## Execution

Shortly after running, the executable encounters an error evaluating JavaScript. Two expected files, *version.json* and *version.plist*, are created but they are empty. Two expected shell scripts, *agent.sh* and *very.sh*, are not created. It thus appears to fail. The same JavaScript error occurs when running as both arm64 and through Rosetta 2 on x86\_64, so the failure is not thought to be due to a translation error.

In Figure 4.8, the UDP stream of *Silver Sparrow*'s installer package can be seen.

```
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env: SHELL=/bin/zsh
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env: MallocSpaceEfficient=0
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env: HOME=/Users/chrles
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env:
__CF_USER_TEXT_ENCODING=0x1F5:0x0:0x0
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env: TMPDIR=/var/folders/nl/
9tjc3k7x10g7gtxq6h3fs83m0000gn/T/
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env:
XPC_SERVICE_NAME=application.com.apple.installer.
1152921500312165612.1152921500312165617
<119>Mar 7 11:24:53 chrless-Mac Installer[2602]: Env: XPC_FLAGS=0x0
<117>Mar 7 11:24:53 chrless-Mac Installer[2602]: Installation Log
<117>Mar 7 11:24:53 chrless-Mac Installer[2602]: Opened from: /Users/chrles/
Downloads/SilverSparrow/update.pkg
<116>Mar 7 11:24:53 chrless-Mac Installer[2602]: Failed to load specified
background image
<119>Mar 7 11:24:57 chrless-Mac Installer[2602]: Product archive /Users/chrles/
Downloads/SilverSparrow/update.pkg trustLevel=350
<119>Mar 7 11:25:00 chrless-Mac Installer[2602]: External component packages (1)
trustLevel=350
<115>Mar 7 11:25:00 chrless-Mac Installer[2602]: Could not load resource readme:
(null)
<115>Mar 7 11:25:00 chrless-Mac Installer[2602]: Could not load resource license:
(null)
<119>Mar 7 11:25:57 chrless-Mac Installer[2602]: JS: Package Authoring Error:
Exception thrown while running installation check. TypeError: null is not an object
(evaluating 'data.args')
<115>Mar 7 11:25:57 chrless-Mac Installer[2602]: Installation checks failed.
<115>Mar 7 11:25:57 chrless-Mac Installer[2602]: Installation check failure.
There was an error reading the package.. An error occurred while evaluating
JavaScript for the package..

Packet 1781. 55 client pkts, 0 server pkts, 0 turns. Click to select.
Entire conversation (5308 bytes) Show data as ASCII Stream 103
```

Figure 4.8. UDP stream for *Silver Sparrow*'s installer package

### 4.3.5. SysJoker

#### *Background*

*SysJoker* was discovered in December 2021. Examining C&C behavior has led to the thought that it's used only in targeted attacks. However, it is also possible its extension implies it is TypeScript to be used as a part of infected software packages. It is a universal binary, and has the ability to execute commands and files based on remote C&C instructions [34].

#### *Execution*

When double clicked normally, the file opens as a playable media file and no infection markers are seen. Persistence is not set, the second binary is not dropped and run, and there is no attempt to connect to its command and control infrastructure. However, after converting it from its extension *.ts* and giving it execution permissions, it is immediately recognized by finder as a universal Mach-O binary. Running that binary then leads to it functioning correctly. The modified binary was also run through Terminal to force launching as x86\_64, and the same functionality was observed. Persistence was set via a launch agent that runs the dropped binary `textitupdateMacOs` at startup. Attempts to connect to a Google drive to generate its *c2* were also observed. Because a signed universal binary is being run on an M1 Mac, the *.ts* version will default to running the arm64 slice if it runs at all. Therefore, despite not knowing the root cause of the inability to function as *.ts*, it can be said it is not an error or lack of functionality with Rosetta 2 translation.

#### 4.4. Summary

The below tables show the macOS security mechanisms that were encountered for both the functional and nonfunctional malware. Their columns differ slightly as a result of the functional malware not encountering Notarization, and the non-functional malware not encountering File Quarantine.

Table 4.1. The functional malware and the protections that were overridden

Malware Sample	File Quarantine	Gatekeeper	Malware Protection
CDDS / MacMa	X		
DazzleSpy	X		
EvilQuest		X	X
FinSpy		X	
Hydromac		X	
Komplex	X		
MacDownloader			X
Realtime-Spy		X	
Ventir		X	
XCSSET			X
ZuRu		X	

Table 4.2. The non-functional malware and the protections that were overridden

Malware Sample	Gatekeeper	Notarization	Malware Protection
AppleJeus / MacLoader	X		X
Crisis	X		
Shlayer		X	X
Silver Sparrow		X	
SysJoker	X		

Table 4.3 covers the functionality that was observed to determine a successful run. The functionality was grouped into 3 columns: Persistence, Networking, and Harvesting. These correspond to the malware ensuring long-term existence on a system, attempting to connect out from a system, or harvesting information from the system, respectively.

Table 4.3. The functional malware and observed high-level behavior

Malware Sample	Persistence	Networking	Harvesting
CDDS / MacMa	X		X
DazzleSpy	X		
EvilQuest	X	?	
FinSpy		X	
Hydromac		X	
Komplex		X	
MacDownloader			X
Realtime-Spy		X	X
Ventir	X		
XCSSET		X	
ZuRu		X	

## 4.5. Discussion

### 4.5.1. Rosetta 2 Translation

From the above results, it is clear that Rosetta 2 is willing to translate x86\_64 malware, even in cases where the sample is unsigned, not notarized, and fingerprinted as malware. It does this translation well, as none of the compatible samples malfunctioned as a result of Rosetta 2 failing to translate properly. However, given that the user consistently had to give permissions, and occasionally override malware protections, it is reasonable that it translated the executables. Rosetta 2 is translating what it is told to translate, so it is functioning correctly. That said, because x86\_64 malware translated through Rosetta 2 does not have to be signed, while arm64 malware does and also requires that all dependencies are ported to arm64, it may be preferable to malware authors to avoid preparing arm64 versions of their programs at all. In most cases with new malware tested, and even with much of the older malware tested, overriding malware protection was not necessary because the binaries were not fingerprinted as malware by *XProtect*, despite some being around for over 5 years. As a result, users usually would only need to be socially engineered into overriding a warning from Gatekeeper, at which point they would be infected. Running unsigned code is a vector that would not be possible without Rosetta 2, since arm64 code

requires a signature to run at all.

#### 4.5.2. Viability of Infection

Most Macs are infected by applications or executables that require user interaction [57]. Malware authors employ a large number of methods to convince users to infect themselves. These methods include but are not limited to: packaging malware with shareware, bundling malware with pirated applications, masquerading as fake applications, spreading through malicious attachments, and using search engine optimization or sponsored links to have malicious web results show highly.

The impact and viability of these samples seems to be lowered due to the general lack of signing and requiring of Rosetta 2 to be installed. However, many users will happily run something anyway when presented with a Gatekeeper warning, especially if any of the above methods are used. Indeed, almost all of the tested malware samples utilized one or more of those techniques for initial infection, many to great success.

#### 4.5.3. Conclusions

Four malware samples were tested that were simply applications. *XCSSET*, *MacDownloader*, and *Realttime-Spy* were the three that were functional. All of them were x86\_64, and completely unsigned. The one that did not work by default on M1, *ZuRu*, was a universal binary and signed. It is not surprising it was signed, since a universal binary would be expected to default to arm64 on an M1. *ZuRu* crashed, however, when run as arm64 because the malicious dynamically linked library it loads in, *libcrypto.2.dylib*, was only built targeting x86\_64. When an app is run as arm64, arm64 is expected to be the architecture for everything else the app needs. If *ZuRu* is executed as x86\_64 instead by manually going into the information for the app and checking “Run with Rosetta”, it runs without crashing. The same Gatekeeper warning occurs whether run as x86\_64 or arm64. The difference is that as just x86\_64, it would have worked without user intervention required to manually specify the proper architecture. This has an interesting implication: life might be easier

for malware staying with x86\_64, rather than building as universal or arm64. Malware authors porting to a universal binary have many considerations to port, including apps, plugins, custom frameworks, static and dynamic libraries, build and command-line tools, launch agents and daemons [3]. In addition, it's not possible to debug the arm64 slice of a universal binary on an Intel Mac. Therefore, if authors do not have an M1, they are essentially building on good faith. Finally, to run the arm64 slice, the program would need to be signed. The downside to x86\_64 only, however, is that it assumes users with M1 Macs will have Rosetta 2 installed or be willing to install it when prompted. When comparing downsides for both, it is reasonable to think that some malware authors will keep building solely for x86\_64 as long as Rosetta 2 lets them.

## Chapter 5. Memory Forensics

### 5.1. Background

Samples were run on a host running macOS Monterey 12.1.0 with networking disabled. Command Line Developer Tools and Rosetta 2 were installed for malware functionality, and Surge 21.12.14 was installed to acquire memory from the host. Analysis was performed with an unreleased version of Volatility 2 with M1 Mac support. Volatility, the new version of it, and efforts to get it functional are discussed in Chapter 6.

### 5.2. Malware chosen

Three samples were chosen for further analysis, all due to observed or purported functionality. *CDDS* was chosen because it was found recently and its harvesting functionality was of interest. *EvilQuest* was chosen because it is also relatively modern, and the infection of other files, killing of processes, and attempted memory-only execution seemed likely to provide good analysis opportunities. It was fingerprinted as malware by *XProtect*, so SIP was disabled on the host so that malware protection could be overridden and the executable run. *Realtime-Spy* was chosen because it is an example of commercial spyware and displayed significant functionality during execution that could be investigated further.

### 5.3. Memory Analysis

The first step was to enumerate the processes in the memory sample and identify any suspicious ones. This step was accomplished through the *mac\_pstree* and *mac\_tasks* Volatility plugins. The *mac\_pstree* plugin not only identifies processes, but also constructs a process tree that shows parent/child relationships between them. The *mac\_tasks* plugin was used to line up processes with metadata like execution time to determine suspicious processes.



```

remnux@remnux:~/aarch64Volatility$ python2 vol.py -f /mnt/images/chapter5/chapter5/cyber2-2.local/20220314185053/memory/data.lime --profile=
MacMonterey_12_1_21C52_arm64_t8101arm64 --shift=265846784 --dtb=34498412544 mac_pstree
Volatility Foundation Volatility Framework 2.6.1
WARNING : volatility.debug : zone_name has no offset in object zone. Check that vtypes has a concrete definition for it.
Name      Pid      Uid
WARNING : volatility.debug : zone_name has no offset in object zone. Check that vtypes has a concrete definition for it.
kernel_task 0        0
..launchd 1        0
..sudo 3349    0
..com.apple.questd 3350    0
..MTLCompilerServi 2845    502
..com.apple.WebKit 2844    502
..MTLCompilerServi 2843    502
..rtcfg 2812    502
..relaunch 2814    502
..LocalAuthenticat 2768    502

```

Figure 5.1. *mac\_pstree* output identifying relationship between *relaunch* and *rtcfg*

With this knowledge, the process tree was used to identify another previously unknown process named *relaunch* that was a child of a suspected malicious process named *rtcfg*. Based on its name, *relaunch* is suspected to be used for persistence. Another process, *com.apple.questd*, was also found and was the Launch Agent that *EvilQuest* loaded in. Using the process identifiers (PIDs) gathered from the previous plugin output, *mac\_lsof* was run next. The *mac\_lsof* plugin lists the open file descriptors of a process. Using this plugin with the malicious PIDs, another clear indicator of compromise was discovered. As shown in Figure 5.2, the *system.rtcfg* folder was discovered. Looking up this directory yields results regarding the *Realtime-Spy* malware and its keylogging capabilities [60]. Such reports also confirm the hypothesis that the *relaunch* process is used for persistence throughout the current session.

```

remnux@remnux:~/aarch64Volatility$ python2 vol.py -f /mnt/images/chapter5/chapter5/cyber2-2.local/20220314185053/memory/data.lime --profile=
MacMonterey_12_1_21C52_arm64_t8101arm64 --shift=265846784 --dtb=34498412544 mac_lsof -p 2812
Volatility Foundation Volatility Framework 2.6.1
WARNING : volatility.debug : zone_name has no offset in object zone. Check that vtypes has a concrete definition for it.
WARNING : volatility.debug : zone_name has no offset in object zone. Check that vtypes has a concrete definition for it.
PID      File Descriptor File Path
-----
2812     0 /Macintosh HD/dev/null
2812     1 /Macintosh HD/dev/null
2812     2 /Macintosh HD/dev/null
2812     3 /Macintosh HD/System/Volumes/Data/Data/Users/user/Library/Caches/system.rtcfg/Cache.db
2812     4 /Macintosh HD/System/Volumes/Data/Data/Users/user/Library/Caches/system.rtcfg/Cache.db-wal
2812     5 /Macintosh HD/System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Versions/A/Resou
rces/HIToolbox.rsrc
2812     6 /Macintosh HD/System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Versions/A/Resou
rces/English.lproj/Localized.rsrc
2812     7 /Macintosh HD/System/Volumes/Data/Data/Library/Application Support/CrashReporter/SubmitDiagInfo.domains
2812     8 /Macintosh HD/System/Volumes/Data/Data/Users/user/Library/Caches/system.rtcfg/Cache.db-shm
2812     9 <netpolicy>
remnux@remnux:~/aarch64Volatility$

```

Figure 5.2. *mac\_lsof* output for known malicious PIDs

An indicator of compromise was found for the *CDDS* malware in a slightly different way. The *mac\_list\_files* plugin, which lists all files in a memory dump, was run against the memory sample and had its output redirected to a file. The list of files was then searched for

Rosetta 2 artifacts that would be generated by the *CDDS* malware, with the understanding that if any hits were found then it would be proven that the malware had been translated and run on the host. Figure 5.3 shows the output of the search, where multiple AOT files were found for malicious binaries associated with *CDDS*. These binaries could be dumped using an updated version of *mac\_procdump* for further analysis. The updates to that plugin and *mac\_procdump* are discussed in detail in Chapter 6.

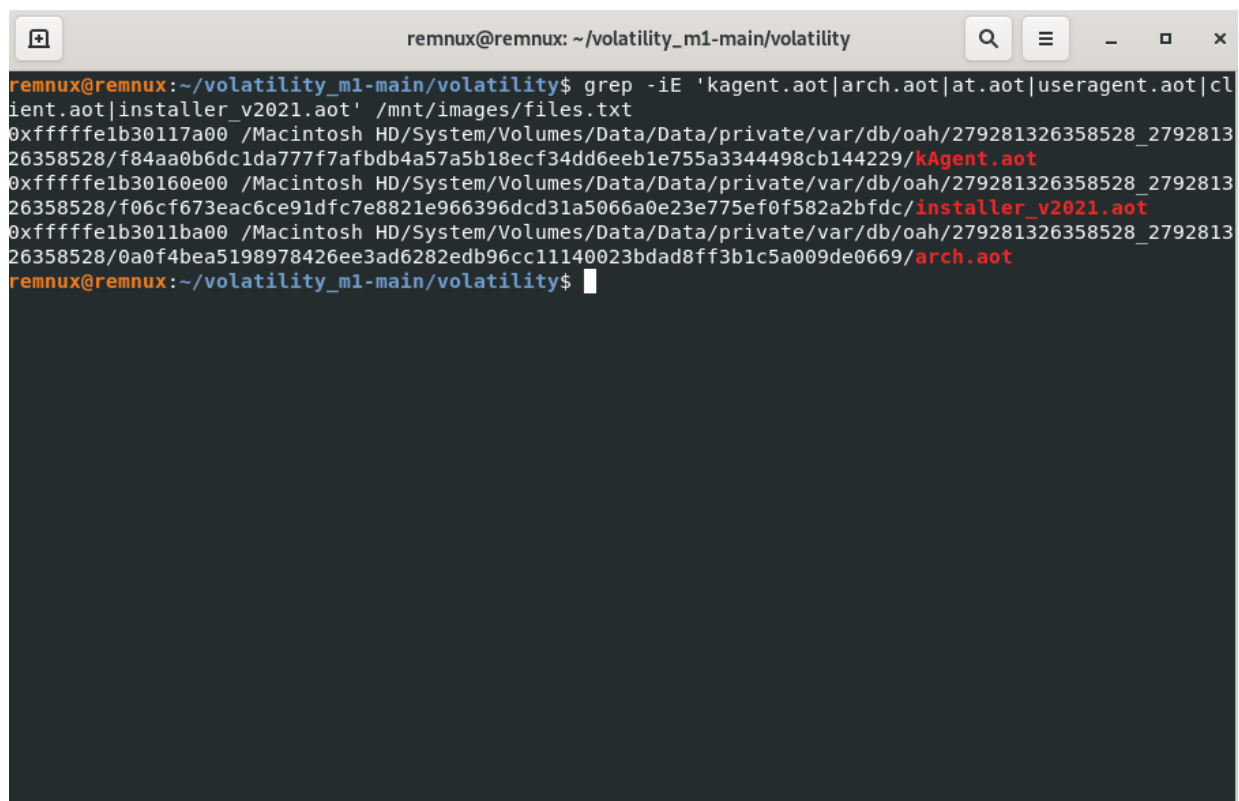
A terminal window titled 'remnux@remnux: ~/volatility\_m1-main/volatility' showing the output of a grep command. The command is 'grep -iE 'kagent.aot|arch.aot|at.aot|useragent.aot|client.aot|installer\_v2021.aot' /mnt/images/files.txt'. The output lists three files with their paths and hashes: 'kAgent.aot', 'installer\_v2021.aot', and 'arch.aot'. The terminal prompt is 'remnux@remnux:~/volatility\_m1-main/volatility\$'.

Figure 5.3. Artifacts from Rosetta 2 translation for the *CDDS* malware sample

The process *CarbonComponentScannerXPC* was investigated but had no official documentation online, and only a few posts on forums asking what it was. Those forums speculated it was related to Rosetta 2, but none confirmed it. Interest was piqued, however, in determining if this process was a part of Rosetta 2, as it potentially relates to the research in this paper. The process was dumped using *mac\_procdump*, then the *strings* [26] util was run on that dumped executable. Figure 5.4 shows some of the many hits for strings related to the Rosetta 2 runtime. This output appears to confirm relation to Rosetta 2.

```
remnux@remnux: ~/volatility_m1-main/volatility
strings -n 8 dumpDir/task.509.0x7ff7ffca1000.dmp | gr
ep -E 'x86|jit|aot|translat|rosetta'
rosetta_version
integer overflow in aot metadata load command fragment list
integer overflow in aot metadata load x86 image path
jit instruction
interpreter jit allocation too small
offsets.x86_instruction_length.has_value()
is_valid_x86_thread
translate_indirect_branch
x86_gpr_state_from_arm_state
expected ARM LR to be in translated code: lr_abi_kind=%hhu arm_pc=0x%llx arm_lr=0x%llx
expected ARM LR to be in translated code: lr_abi_kind=%hhu arm_pc=0x%llx _saved_lr=0x%llx
expected next ARM PC to be in translated code: abi_kind=%hhu new_arm_pc=0x%llx
lr_abi_info.u.translated_code.instruction_extents.kind == InstructionOffsetKind::Syscall
expected saved LR to be in translated code: lr_abi_kind=%hhu saved_lr=0x%llx arm_pc=0x%llx arm_lr=0x
%llx
x86_rip.has_value()
did not find an x86 pc. abi_kind=%hhu arm_pc=0x%llx arm_lr=0x%llx
emulated forward to an arm pc that isn't in translated code. arm_pc=0x%llu abi_kind=%hhu emulation
interval=[0x%llu,0x%llu) instruction_interval=[0x%llu, 0x%llu) x86_rip=0x%llu
x86_fpr_state_from_arm_state
arm_gpr_state_from_x86_state
arm_fpr_state_from_x86_state
find_translation_in_tree_x86
metadata->x86_base_address <= x86_address
iterate_fragments_in_x86_interval_holding_lock
set_jit_page_permission
mprotect failed on jit write fault: %lld
handle_jit_breakpoint
```

Figure 5.4. *mac\_procdump* output for the *CarbonComponentScannerXPC* process

## 5.4. Analysis Discussion

As proven by the above analysis, the chosen samples are indeed functioning and running. Indicators of compromise were found for each of the three samples analyzed. Therefore, memory forensics has provided an avenue for confirmation that Rosetta 2 translates malicious x86\_64 programs. Using Volatility, indicators of compromise such as running processes, open file descriptors, Rosetta 2 artifacts, and strings from dumped processes were plainly seen. Memory forensics was extremely useful in finding indication these samples ran, and even information about a process unrelated to malware, allowing for insights that would often not otherwise be possible. These analysis capabilities would not exist without the prior contributions made to get Volatility working on M1 Macs.

## Chapter 6. Other Contributions

### 6.1. Volatility

Volatility is the defacto open-source framework that analysts utilize to perform memory forensics [64]. Through the use of plugins and OS-specific profiles that contain symbol and type information for the kernel version, data from a memory sample is parsed and extracted before being presented to the user. Volatility offers fairly robust support for Windows, macOS, and Linux. However, when Apple released M1 Macs, Volatility had no immediate way to support analysis of them, as there was more work to be done than just using Apple's kernel debug kits to create a new profile [42].

### 6.2. Volatility on M1 Macs

An address space, which is used by Volatility for address translation and mimics the memory management unit of the CPU, for arm64 did not exist prior to M1. One would be necessary for Volatility to support M1 memory forensics, however. An arm64 address space was made by Tsahi Zidenberg and pushed to Github [67], and this was used as the basis for M1 address translation.

Next, Volatility requires knowledge of the location of the kernel page tables and the value of the kernel address space layout randomization. The former is used for address translation and the latter is necessary because the offset provided in the Volatility profile will not be the same as the actual offset in memory. These values could be automatically obtained on Intel Macs, but are currently necessary for an analyst to find themselves via the metadata of a memory acquisition tool such as Surge.

With the above fixes and one more fix for userland virtual address translation, plugins could now be tested. Plugins were tested in bulk and output of each plugin was studied. There were many plugins that required modifications to become functional on M1. Some fixes were as simple as adding arm64 architecture to the list of valid architectures for the

plugins to check. There were also multiple plugins that relied on an Intel-specific data structure to determine information about a process. These plugins now use a new API that is functional on both architectures. A final breaking change was kernel symbols and types that were moved or were missing from M1 Macs. Multiple plugins had equivalent symbols or work-arounds that led to the same functionality. Some, however, currently have not been able to be updated for M1 as a result of missing symbols. Overall, almost all Volatility plugins are now fully functional on M1.

### 6.3. Rosetta 2 Analysis and Plugins

Rosetta 2 technical analysis began by writing a test program that generated artifacts in memory and had a dependency for a test library. The program was then run, and the memory mappings of the running processes were viewed through the *vmmap* macOS command. Through the memory mappings it could be seen that there were two mappings associated with each process, including the test library. The *mac-procdump* and *mac-librarydump* plugins were tested and only grabbed the Intel executables, not the second mapping for the AOT artifacts for each. Both plugins were updated to be aware of and extract the AOT artifacts, adding valuable artifacts that investigators would now be aware of. These artifacts are particularly helpful, as they are proof of Intel binaries being executed on an M1. It is also significant in that these artifacts persist through reboots, even if the original executables are deleted.

REGION TYPE	START - END	[ VSIZE	RSBNT	DIRTY	SWAP]	PRT/MAX	SHRMOD	REGION DETAIL
__TEXT	102281000-102282000	[ 4K	4K	0K	0K]	r-x/r-x	SM=COW	*/test_program
__DATA_CONST	102282000-102283000	[ 4K	4K	4K	0K]	r--/rw-	SM=COW	*/test_program
__LINKEDIT	102284000-102285000	[ 4K	4K	0K	0K]	r--/r--	SM=COW	*/test_program
mapped file	102285000-102287000	[ 8K	8K	0K	0K]	r-x/rwx	SM=COW	/private/var/db/*/test_program.aot
mapped file	102287000-10228b000	[ 16K	16K	0K	0K]	r-x/r-x	SM=COW	/usr/libexec/rosetta/runtime
mapped file	10228c000-10228d000	[ 4K	4K	0K	0K]	r--/rwx	SM=COW	/private/var/db/*/test_program.aot
mapped file	102295000-102299000	[ 16K	16K	0K	0K]	r-x/r-x	SM=COW	/usr/libexec/rosetta/runtime
__TEXT	10a297000-10a2eb000	[ 336K	320K	0K	0K]	r-x/rwx	SM=COW	*/libRosettaRuntime
Rosetta Thread Context	10a2fc000-10a2fd000	[ 4K	0K	0K	0K]	---/rwx	SM=NUL	
Rosetta Return Stack	10a301000-10a302000	[ 4K	0K	0K	0K]	---/rwx	SM=NUL	
Rosetta Generic	10a306000-10a307000	[ 4K	0K	0K	0K]	---/rwx	SM=NUL	
__TEXT	10a8ae000-10a8af000	[ 4K	4K	0K	0K]	r-x/rwx	SM=COW	*/test_library.dylib
__DATA_CONST	10a8af000-10a8b0000	[ 4K	4K	4K	0K]	r--/rwx	SM=COW	*/test_library.dylib
__LINKEDIT	10a8b1000-10a8b2000	[ 4K	4K	0K	0K]	r--/rwx	SM=COW	*/test_library.dylib
mapped file	10a8b2000-10a8b4000	[ 8K	8K	0K	0K]	r-x/rwx	SM=COW	/private/var/db/*/test_library.dylib.aot
mapped file	10a8b4000-10a8b8000	[ 16K	16K	0K	0K]	r-x/r-x	SM=COW	/usr/libexec/rosetta/runtime
mapped file	10a8b9000-10a8ba000	[ 4K	4K	0K	0K]	r--/rwx	SM=COW	/private/var/db/*/test_library.dylib.aot

Figure 6.1. *vmmap* showing dual mappings for the test program and library

A new plugin, *mac\_rosetta*, was also developed which automatically extracts and lists all AOT files. Figure 6.2 shows the output of this plugin against the sample program and library. The extraction of AOT files could theoretically be done on a live machine, but the folder that contains them is protected by SIP. On a live machine, therefore, the only way to access that folder is by disabling SIP, which involves rebooting the machine and loosening restrictions. After disabling SIP, a user would then manually navigate into that folder to grab AOT files for analysis. While reasonable to do on an individual level, that approach quickly becomes infeasible as a part of live incident response with many potential machines involved. These enhanced and new plugins, as well as all plugins updated to be functional for M1 Macs, represent a significant leap in the ability to fully forensically analyze an M1 Mac.

Pid	Name	Base	AOT Path
417	CarbonComponentS	0x0000000102fa0000	Data/private/var/db/oah/[snip]/CarbonComponentScannerXPC.aot
2277	test_program	0x0000000102285000	Data/private/var/db/oah/[snip]/test_program.aot
2277	test_program	0x000000010a8b2000	Data/private/var/db/oah/[snip]/test_library.dylib.aot
2277	test_program	0x00000002025ed000	Data/private/var/db/oah/[snip]/dyld.aot

Figure 6.2. *mac\_rosetta* extracting AOT artifacts

## Chapter 7. Conclusion and Future Work

### 7.1. Conclusion

Apple made the decision to develop Rosetta 2 to ease the transition from Intel architecture to M1 architecture. A side effect of this decision was that all previous x86\_64 malware would run on their new machines. This research explored the consequences of this reality by determining the functionality of malicious x86\_64 binaries on M1 Macs and the viability of continuing to run them over arm64 versions. Functionality was explored through two angles: high-level dynamic malware analysis and memory forensics. Both angles had their results thoroughly analyzed and discussed. Finally, the research effort to bring memory forensics to M1 was detailed, as that made the memory forensics in Chapter 5 possible.

### 7.2. Future Work

One constraint to the analysis performed in Chapter 4 was that connectivity checks were not bypassed. Getting around these checks would be possible either by debugging the process at the instruction level and skipping over the checks, or by using a program like INetSim [8] that will return 200's for any connection attempt. Either method would have allowed further functionality to be observed for multiple of the samples tested.

Another contribution would be to investigate more samples on M1 using memory forensics. As there were only three samples on which memory forensics was performed, there is potentially much more to be learned through continued analysis of a wider variety of samples.

A research effort that is happening concurrently is documenting APIs commonly used by malware on Intel, and determining whether or not the same functionality exists on M1. This effort is taking a deeper dive into the exposed macOS functionality and is resulting in a better understanding of how malware will need to function to achieve its goals on M1 devices. As a result, these APIs will be better understood, defended against, and identified

in digital forensics investigations.

The research in this paper was also performed using Volatility 2 because of its wider adoption. Porting the efforts detailed in Chapter 6 to Volatility 3 [4] will be necessary as it starts to become more used over Volatility 2.



## References

- [1] About the rosetta translation environment. [Online]. Available: <https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment>
- [2] arch(1) [osx man page]. [Online]. Available: <https://www.unix.com/man-page/osx/1/arch/>
- [3] Building a universal macos binary. [Online]. Available: <https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary>
- [4] Changes between volatility 2 and volatility 3. [Online]. Available: <https://volatility3.readthedocs.io/en/latest/vol2to3.html>
- [5] Creating launch daemons and agents. [Online]. Available: <https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html>
- [6] How much does a data breach cost? [Online]. Available: <https://www.ibm.com/security/data-breach>
- [7] ikittens: Iranian actor resurfaces with malware for mac (macdownloader). [Online]. Available: <https://iranthreats.github.io/resources/macdownloader-macos-malware/>
- [8] Inetsim manpage. [Online]. Available: <https://manpages.debian.org/testing/inetsim/inetsim.1.en.html>
- [9] Installing rosetta. [Online]. Available: <https://support.apple.com/en-us/HT211861>
- [10] A launchd tutorial. [Online]. Available: <https://www.launchd.info/>
- [11] Lime - linux memory extractor. [Online]. Available: <https://github.com/504ensicsLabs/LiME>
- [12] Lsarchitecturepriority documentation. [Online]. Available: [https://developer.apple.com/documentation/bundleresources/information\\_property\\_list/lsarchitecturepriority](https://developer.apple.com/documentation/bundleresources/information_property_list/lsarchitecturepriority)
- [13] The mac security blog. [Online]. Available: <https://www.intego.com/mac-security-blog/>
- [14] Mach-o architecture. [Online]. Available: [https://developer.apple.com/documentation/foundation/1495005-mach-o\\_architecture](https://developer.apple.com/documentation/foundation/1495005-mach-o_architecture)
- [15] Malware from objective-see.com. [Online]. Available: <https://objective-see.com/malware.html>
- [16] Malwarebytes labs. [Online]. Available: [blog.malwarebytes.com](http://blog.malwarebytes.com)
- [17] Notarizing macos software before distribution. [Online]. Available: [https://developer.apple.com/documentation/security/notarizing\\_macos\\_software\\_before\\_distribution](https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution)

- [18] Objective by the sea - the mac security conference. [Online]. Available: <https://objectivebythesea.com/>
- [19] Objective-see. [Online]. Available: <https://objective-see.com>
- [20] Overview of the mach-o executable format. [Online]. Available: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html>
- [21] Protecting against malware in macos. [Online]. Available: <https://support.apple.com/guide/security/protecting-against-malware-sec469d47bd8/web>
- [22] Reverse engineering - understanding and breaking stuff for fun and profit! [Online]. Available: <https://reverse.put.as>
- [23] Rosetta 2 on a mac with apple silicon. [Online]. Available: <https://support.apple.com/guide/security/rosetta-2-on-a-mac-with-apple-silicon-secebb113be1/web>
- [24] Security integrity protection. [Online]. Available: [https://developer.apple.com/documentation/security/disabling\\_and\\_enabling\\_system\\_integrity\\_protection](https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection)
- [25] Sentinelone blog. [Online]. Available: <https://www.sentinelone.com/blog/>
- [26] strings(1) — linux manual page. [Online]. Available: <https://man7.org/linux/man-pages/man1/strings.1.html>
- [27] Volexity. [Online]. Available: <https://www.volexity.com>
- [28] Volexity blog. [Online]. Available: <https://www.volexity.com/blog/>
- [29] The xcsset malware: Inserts malicious code into xcode projects, performs uxss backdoor planting in safari, and leverages two zero-day exploits. [Online]. Available: [https://documents.trendmicro.com/assets/pdf/XCSSET\\_Technical\\_Brief.pdf](https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf)
- [30] Amanda Gorton. (2021) How we ported linux to the m1. [Online]. Available: <https://www.corellium.com/blog/linux-m1>
- [31] Amnesty International. (2020) German-made finspy spyware found in egypt, and mac and linux versions revealed. [Online]. Available: <https://www.amnesty.org/en/latest/research/2020/09/german-made-finspy-spyware-found-in-egypt-and-mac-and-linux-versions-revealed/>
- [32] Andrew Case. (2012) Mac memory analysis with volatility. [Online]. Available: <https://www.slideshare.net/AndrewDFIR/mac-memory-analysis-with-volatility>
- [33] Andrew Case, Ryan D. Maggio, Modhuparna Manna, Golden G. Richard III. (2020) Memory analysis of macos page queues. [Online]. Available: <https://doi.org/10.1016/j.fsidi.2020.301004>

- [34] Avigayil Mechtinger, Ryan Robinson, Nicole Fishbein. (2022) New SysJoker Backdoor Targets Windows, Linux, and macOS. [Online]. Available: <https://www.intezer.com/blog/malware-analysis/new-backdoor-sysjoker/>
- [35] Cedric Owens. (2021) macos gatekeeper bypass (2021 edition). [Online]. Available: <https://cedowens.medium.com/macos-gatekeeper-bypass-2021-edition-5256a2955508>
- [36] Dani Creus, Tyler Halfpop, Robert Falcone. (2016) Sofacy’s ‘komplex’ os x trojan. [Online]. Available: <http://researchcenter.paloaltonetworks.com/2016/09/unit42-sofacys-komplex-os-x-trojan/>
- [37] Erye Hernandez. (2021) Analyzing a watering hole campaign using macos exploits. [Online]. Available: <https://blog.google/threat-analysis-group/analyzing-watering-hole-campaign-using-macos-exploits/>
- [38] Global Research Analysis Team, Kaspersky Lab. (2018) Operation applejeus: Lazarus hits cryptocurrency exchange with fake installer and macos malware. [Online]. Available: <https://securelist.com/operation-applejeus/87553/>
- [39] hoakley. (2020) How notarization works. [Online]. Available: <https://eclecticlight.co/2020/08/28/how-notarization-works/>
- [40] hoakley. (2021) How rosetta complicates call chains on m1 macs. [Online]. Available: <https://eclecticlight.co/2021/01/27/how-rosetta-complicates-call-chains-on-m1-macs/>
- [41] Jaron Bradley. (2021) Shlayer malware abusing gatekeeper bypass on macos. [Online]. Available: <https://www.jamf.com/blog/shlayer-malware-abusing-gatekeeper-bypass-on-macos/>
- [42] Joshua Duke. (2021) Memory forensics comparison of apple m1 and intel architecture using volatility framework. [Online]. Available: [https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=6527&context=gradschool\\_theses](https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=6527&context=gradschool_theses)
- [43] Joshua Long. (2020) Mac malware exposed: Xcsset, an advanced new threat. [Online]. Available: <https://www.intego.com/mac-security-blog/mac-malware-exposed-xcsset-an-advanced-new-threat/>
- [44] Koh M. Nakagawa. (2019) Reverse-engineering rosetta 2. [Online]. Available: <https://ffri.github.io/ProjectChampollion/part1/>
- [45] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. John Wiley & Sons, 2014.
- [46] lordx64. (2021) Osx/hydromac: A new macos malware leaked from a flashcards app. [Online]. Available: <https://blog.confiant.com/osx-hydromac-a-new-macos-malware-leaked-from-a-flashcards-app-2af28f1caa9e>

- [47] Lysa Myers. (2012) More on osx/crisis — advanced spy tool. [Online]. Available: <https://www.intego.com/mac-security-blog/more-on-osxcrisis-advanced-spy-tool/>
- [48] Magnet Forensics. Magnet ram capture. [Online]. Available: <https://www.magnetforensics.com/resources/magnet-ram-capture/>
- [49] Marc-Etienne M.Léveillé, Anton Cherepanov. (2022) Watering hole deploys new macos malware, dazzlespy, in asia. [Online]. Available: <https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/>
- [50] Mikhail Kuzin. (2014) The ventir trojan: assemble your macos spy. [Online]. Available: <https://securelist.com/the-ventir-trojan-assemble-your-macos-spy/67267/>
- [51] Modhuparna Manna, AndrewCase, Aisha Ali-Gombe, Golden G.Richard III. (2021) Modern macos userland runtime analysis. [Online]. Available: <https://doi.org/10.1016/j.fsidi.2021.301221>
- [52] Noora Hyvärinen. (2018) Spam campaign targets exodus mac users. [Online]. Available: <https://blog.f-secure.com/spam-campaign-targets-exodus-mac-users/>
- [53] osxreverser. (2020) The finfisher tales, chapter 1: The dropper. [Online]. Available: <https://reverse.put.as/2020/09/26/the-finisher-tales-chapter-1/>
- [54] Parallels. (2020) Install macos monterey 12 virtual machine on a mac with apple m1 chips. [Online]. Available: <https://kb.parallels.com/125561>
- [55] Parallels. (2020) Network modes in parallels desktop for mac. [Online]. Available: <https://kb.parallels.com/4948>
- [56] Patrick Wardle. (2020) Osx.evilquest uncovered. [Online]. Available: [https://objective-see.com/blog/blog\\_0x59.html](https://objective-see.com/blog/blog_0x59.html)
- [57] Patrick Wardle. (2021) All your macs are belong to us. [Online]. Available: [https://objective-see.com/blog/blog\\_0x64.html](https://objective-see.com/blog/blog_0x64.html)
- [58] Patrick Wardle. (2021) Arm'd & dangerous, malicious code, now native on apple silicon. [Online]. Available: [https://objective-see.com/blog/blog\\_0x62.html](https://objective-see.com/blog/blog_0x62.html)
- [59] Patrick Wardle. (2021) Made in china: Osx.zuru. [Online]. Available: [https://objective-see.com/blog/blog\\_0x66.html](https://objective-see.com/blog/blog_0x66.html)
- [60] Phil Stokes. (2018) The dangers of a fake macos cryptowallet keylogger. [Online]. Available: <https://www.sentinelone.com/blog/macos-spyware-dangers-fake-cryptowallet-keylogger/>
- [61] SonicWall. (2022) 2022 sonicwall cyber threat report. [Online]. Available: <https://www.sonicwall.com/2022-cyber-threat-report/>
- [62] Stan Skowronek. (2019) Reverse engineering the m1. [Online]. Available: <https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Reverse-Engineering-The-M1.pdf>

- [63] The Volatility Foundation. (2020) About the volatility foundation. [Online]. Available: <https://www.volatilityfoundation.org/about>
- [64] The Volatility Foundation. (2020) Wiki home. [Online]. Available: <https://github.com/volatilityfoundation/volatility/wiki>
- [65] Thomas Reed. (2021) Mac detections by the numbers. [Online]. Available: [https://objectivebythesea.com/v4/talks/OBTS\\_v4.tReed.pdf](https://objectivebythesea.com/v4/talks/OBTS_v4.tReed.pdf)
- [66] Tony Lambert. (2021) Clipping silver sparrow's wings: Outing macos malware before it takes flight. [Online]. Available: <https://redcanary.com/blog/clipping-silver-sparrows-wings/>
- [67] Tsahi Zidenberg. (2020) Volatility arm64. [Online]. Available: <https://github.com/tsahee/volatility/tree/arm64>
- [68] Volexity. Surge collect pro. [Online]. Available: <https://www.volexity.com/products-overview/surge/>
- [69] P. Wardle, *The Art of Mac Malware*. [Online]. Available: <https://taomm.org/>

## **Vita**

Charles Elliot Glass was born in New Orleans, Louisiana. He graduated with a B.S. in Computer Science from Louisiana State University (LSU) in 2019. After receiving the CyberCorps Scholarship for Service scholarship, he returned to LSU to earn an M.S. in Computer Science. He will graduate with his master's May 2022.